MINING SPATIO-TEMPORAL INFORMATION SYSTEMS

Edited by
Roy Ladner
Kevin Shaw
Mahdi Abdelguerfi

Kluwer Academic Publishers

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
15-MAY-2002	Book Chapter	

4. TITLE AND SUBTITLE
Efficient Storage Of Large Volume Spatial And Temporal Point-Data In An Object-Oriented Database

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
DAVID V. OLIVIER ROY VICTOR LADNER FRANK P. McCREEDY RUTH ANNE WILSON

5d. PROJECT NUMBER
0602435N & 63782N

5e. TASK NUMBER

5f. WORK UNIT NUMBER
74-6731-02,747960-D1

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Research Laboratory
Marine Geoscience Division
Stennis Space Center, MS 39529-5004

8. REPORTING ORGANIZATION REPORT NUMBER
NRL/BC/7440--02-1001

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
NRL
Code 7440.2
SSC MS 3952

ONR
800 N. Quincy Steet
Arlington, VA 22217

10. SPONSOR/MONITOR'S ACRONYM(S)
NRL & ONR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release,distribution is unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Data mining applications must deal with large volumes of data. In particular, Spatio-Temporal Information Systems must efficiently store and access potentially very large quantities of spatial and temporal data. Therefore, storing the data in an efficient and useful way is of great importance. Binary Large Objects (BLOBs) are found in many database systems and have been extensively used in typical database applications for the storage of large volume data. In this chapter, we describe the extension of basic BLOBs for specialized use with spatial and temporal data. These new repositories, Spatial BLOBs and Temporal BLOBs, add additional functionality for the query and retrieval of the repository's contents in a semantically meaningful, object-oriented form. The repositories are designed as flexible frameworks, decoupled from the particular binary format of their internal contents. Custom plug-ins allow the frameworks to be extended to use a particular binary format that is most appropriate for a given data type.

15. SUBJECT TERMS
spatial data, temporal data, data storage, binary large objects, BLOBs

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			David Olivier
unclassified	unclassified	unclassified	Unlimited	19	19b. TELEPHONE NUMBER *(Include area code)* 228-688-5920

Chapter 3

Efficient Storage of Large Volume Spatial and Temporal Point-Data in an Object-Oriented Database

David Olivier, Roy Ladner, Frank McCreedy, Ruth Wilson
Naval Research Laboratory

Key words: spatial data, temporal data, data storage, binary large objects, BLOBs

Abstract: Data mining applications must deal with large volumes of data. In particular, Spatio-Temporal Information Systems must efficiently store and access potentially very large quantities of spatial and temporal data. Therefore, storing the data in an efficient and useful way is of great importance. Binary Large Objects (BLOBs) are found in many database systems and have been extensively used in typical database applications for the storage of large volume data. In this chapter, we describe the extension of basic BLOBs for specialized use with spatial and temporal data. These new repositories, Spatial BLOBs and Temporal BLOBs, add additional functionality for the query and retrieval of the repository's contents in a semantically meaningful, object-oriented form. The repositories are designed as flexible frameworks, decoupled from the particular binary format of their internal contents. Custom plug-ins allow the frameworks to be extended to use a particular binary format that is most appropriate for a given data type.

1. INTRODUCTION

To describe the dynamic state of the environment and extrapolate meaningful knowledge regarding it, Spatio-Temporal Information Systems must store and access large volumes of data. There are three principal reasons for this.

1. The first is that the amount of data required to capture the state of the environment at a given moment in time is large. For example, meteorological data sets describe the state of atmosphere by describing its state at a large number of discrete points. A typical data set might describe the value of some parameter at tens or hundreds of thousands of

points. In turn, many data sets for many different parameters may be required to sufficiently describe the state of the atmosphere at a single moment.

2. The second is that the amount of data may grow rapidly. While all geospatial features change over time, this change is often sufficiently small for a given duration of interest that it can be treated as static, and, therefore, only a single "current" set of conditions is stored. With dynamic data, however, we are typically not only interested in conditions at a single, current moment but at a range of times extending back into the past and possibly forward into the future (predicted conditions). As a result we must store many consecutive snapshots of the environment.
3. The third reason has less to do with what the data is describing and more to do with *how* the data is to be used. If the data is to be used as input for data mining, the data sets must necessarily be large to extract meaningful results.

The expected quantities of spatio-temporal data require that we adopt compact mechanisms for their storage. However, simply storing large quantities of data in a small amount of space does not guarantee that we have met our objectives. The data must be *usable* and, therefore, easily accessible. We can define several criteria for usability:

- *The ability to query for data by semantically meaningful criteria*: In the case of spatio-temporal data this means querying for data by spatial or temporal attributes.
- *The ability to retrieve data in a semantically meaningful form*: This means that the data retrieved must be structured in a meaningful way easily usable by the STIS.
- *Efficiency*: Since the STIS must access large quantities of data the transactions must occur quickly.

These criteria guarantee that the system will be able to effectively use the data it stores. The Spatial and Temporal BLOBs described here were developed for the storage of meteorological, oceanographic, and observational data and were designed with these criteria in mind. They were developed for use in the Geospatial Information Database (GIDB™) System, an STIS developed by the Digital Mapping, Charting and Geodesy Analysis Program (DMAP) of the Naval Research Laboratory [NRL 2002].

2. THE GIDB SYSTEM

The GIDB System is a distributed geospatial mapping application composed of three principal components: a client user-interface for requesting and viewing data products, a central data portal which coordinates

communication between the client and various databases, and an object-oriented database system for the persistent storage of environmental data. The goal of the GIDB System is to assimilate a wide variety of data types from disparate sources into a single data model and view, thereby giving the user a powerful data analysis and planning tool.

The GIDB client application allows the user to access, query, and visualize data. Data may be searched by area-of-interest. Heterogeneous data sets may be viewed in a unified environment. The principal view is a two-dimensional, GIS-like rendering, but data may also be viewed in three dimensions. Additional functionality allows for the querying and viewing of spatially indexed multimedia data.

The GIDB client does not communicate directly with any databases but, instead, makes requests and receives products via the GIDB portal. The portal allows the client to connect not only to the GIDB database but, also, to a wide variety of other databases, both local and remote. All communications between the client and portal occur in a common data transfer format. The portal, however, can communicate with external data sources in their native format. The portal can be extended to communicate with new types of data sources by adding new drivers that handle translations between GIDB's data transfer format and that of the new data source. This extensibility allows a wide variety of data from other sources to be viewed together with data from the GIDB database.

The GIDB database component defines a generic data model facilitating the internal representation of a wide variety of geospatial information. A hierarchic data model organizes data according to scale, thematic layers, and feature classification types. It is implemented using the Java, open source, object-oriented database management system, Ozone [Ozone 2002]. The Spatial and Temporal BLOBs described here have been developed for use within Ozone.

3. THE PROBLEM DOMAIN

Let us examine in more detail exactly what sort of data we intend to store in the Spatial and Temporal BLOBs. Spatio-temporal data sets, in general, might describe entities of a wide variety of shapes (points, lines, polygons, etc.) However, Spatial and Temporal BLOBs were developed specifically to store meteorological, oceanographic, and observational data, and these data sets typically only describe conditions at specific points within the atmosphere or ocean. Therefore, we can make a simplifying assumption: *Spatial and Temporal BLOBs will only model point data.* This allows us to

handle the vast majority of data we are interested in but reduces the complexity of the solution.

Spatial and Temporal BLOBs were each designed to handle a specific subset of the anticipated data. Spatial BLOBs were developed with the specific aim of storing typical meteorological and oceanographic data sets and survey data. These data sets describe multiple points over some area. They fall into two main subcategories:

1. Gridded: Most frequently this type of data is output from a meteorological or oceanographic model. However, it could be any spatial point data where the points are arranged in a horizontal grid.
2. Irregular: The points have no predictable ordering. Typically, this might be observations from multiple weather stations at a given time or samplings from a survey.

Temporal BLOBs were developed to address a different sort of data, specifically observations at a single point over a span of time. One could be used, for example, for storing a series of readings from a weather station or buoy. As an analog to gridded and irregular spatial data, the temporal observations might be regularly or irregularly spaced in time.

Spatial and Temporal BLOBs both describe multiple space-time points, but each holds some dimensions constant while varying for others. A Spatial BLOB describes points at a fixed time but varying positions (in a horizontal plane). Temporal BLOBs describe a point at a fixed position but varying times. Beyond this, however, a wide variety of data configurations exist that we will want to handle (individual data points may have a single attribute or many; all points in a set may have the same or different attributes; the attributes may be of many different types, etc.)

4. AN OBJECT-ORIENTED SOLUTION

Before discussing our solution in more detail, we will examine the particular issues related to implementing a solution in an object-oriented context, both the benefits and drawbacks. Using an object-oriented data model allows for a semantically rich representation of the data coupled with convenient functionality. Using an object-oriented database allows for seamless data storage and retrieval without any need to explicitly marshal and unmarshal the data to and from a tabular or binary format. However, the volume of dynamic spatio-temporal data is often much larger than that of static spatial data and a straightforward extension of the existing object-model to incorporate spatio-temporal data incurs prohibitive performance costs. The goal of this work is to develop a solution for the storage of

spatio-temporal data that resolves these issues while seamlessly integrating into the existing object-oriented system.

On the surface it would seem that there is no reason why GIDB's existing object model cannot be extended to describe large point-data sets. To do this one would simply represent each point in the set as a point object. Suppose a data set describes rain rates over some area. Model each sample as a point object with an associated position and rain rate value. This approach is straightforward, integrating nicely with the existing system, and it would be viable for small data sets. However, when applied to realistic volumes of data it quickly becomes untenable. We will refer to it as the naïve object approach and will examine its shortcomings in more detail.

Applying the naïve object approach to high volumes of point-data results in two performance problems:

1. Storage: A persistent object representation of a point takes significantly more disk space than a straight binary representation of the same data. For a large number of points this problem becomes critical.
2. Indexing: The existing GIDB system uses an R*-tree to spatially index features [Owen 1997]. The implementation in use is effective for indexing a moderate number (tens of thousands) of objects. However, as the number of points in the index grows, the overall query time increases. Eventually, the index cannot handle the number of points and fails (on a PC with 500 megabytes of RAM, the maximum was between four-hundred and five-hundred thousand entries). Typical spatio-temporal data sets could quickly overwhelm such an index.

Of course, many applications do exist which store large point-data sets compactly. These store the data in binary files. Each file typically describes a single data set. It contains many individual byte records describing the relevant conditions at a point within the data set. In general, a strictly binary representation of data is significantly smaller than that same data stored in object form.

While the binary file approach is not, in itself, desirable for use in the GIDB context (we lose the advantages of seamless persistence of data), it points us in the direction we want to go. The solutions we will describe here are object-binary hybrids, repository objects that internally stores the data in binary format but provide a purely object-oriented interface to the rest of the system. These solutions are developed on the foundation of Binary Large Objects (BLOBs) which are found in many database systems but add functionality for spatially and temporally querying for the data and translating it from binary to object form.

5. REQUIREMENTS

Before describing Spatial and Temporal BLOBs in detail we will enumerate the requirements motivating their design. These requirements fall into three categories:

1. *Functionality*: The solution must seamlessly integrate with the existing object-oriented system, allowing for the data to be queried by spatial or temporal criteria (area-of-interest or duration-of-interest), and returned in a semantically meaningful object form.
2. *Performance*: The solution must optimize for three criteria:
   Storage size: The size of the stored data is very important. We will seek to minimize use of disk space.
   Indexing: This is the criterion that decisively ruled out the naive object approach. We must use a solution that will not populate the existing indexing scheme with hundreds of thousands of objects.
   Speed: Methods to optimize for other performance criteria cannot be used at the expense of excessive data access times. In an interactive application reasonable response times are essential to usability.
3. *Flexibility*: The solution must work for a wide variety of spatio-temporal data types.

6. TOWARDS A SOLUTION

The design of the Spatial and Temporal BLOBs is based on several key design decisions that allow for the successful resolution of our requirements.

Wrap the data set as a single object. Instead of defining each data point as an object in our model, we will develop a repository object that wraps all of the contents of a data set into a single object. By changing the correspondence from

data point → single object feature

to

data set → single object feature

we resolve the problem of overwhelming the index. The primary database index has a single object added, not hundreds of thousands, so the effect on the performance of the system as a whole is negligible (although the repository must still internally index the data).

Within the repository, represent the data in binary format. By wrapping a data set into a single object we have gained complete freedom in how this

data is internally represented. By storing it in binary we obtain a significant reduction in storage size.

Spatially or temporally index the contents of the repository. The repository will receive requests for data that matches a given area-of-interest or duration-of-interest. To efficiently retrieve only the relevant data the repository must have an internal spatial or temporal index of its contents.

Provide conversion functionality. We require that the repository seamlessly integrates with the existing object-oriented system. As a consequence, it cannot respond to requests for data by returning it in binary format. Therefore, the repository must have functionality to convert from the data's internal binary representation to an object representation suitable for return to the larger system.

Define an abstract framework. Deciding to internally represent the data in binary form is an excellent way to reduce storage size, but it leaves us with the question, "What binary format should be used?" Different formats are best suited for different types of data. For example, in the case of spatial data, a format that is well suited for gridded data may not work with irregular data. Typically file formats for gridded data do not store the position of each point in the corresponding record. Instead they store header information describing the grid that allows the position of each point to be computed. On the other hand, file formats which are intended to describe irregular data must store the position of each point since there is no way of inferring it. While the format for the storage of irregular data is sufficiently general to describe gridded data, the storage cost of storing position information for each point is unnecessary. Since optimized storage size is a central priority and much of the data we will handle is gridded, this is not a cost we are willing to incur. Our decision is to leave the format unspecified. Instead the Spatial and Temporal BLOBs are abstract frameworks that can be tailored to allow for different internal binary representations best suited to the data type they will contain.

7. THE DESIGN

We have outlined the general design decisions that our solution will implement. Let us see how these decisions will translate into a high level design[1].

[1] The Spatial and Temporal BLOBs have essentially the same design. In this and the following section, for convenience, we will refer to them collectively as the Spatio-Temporal BLOB.

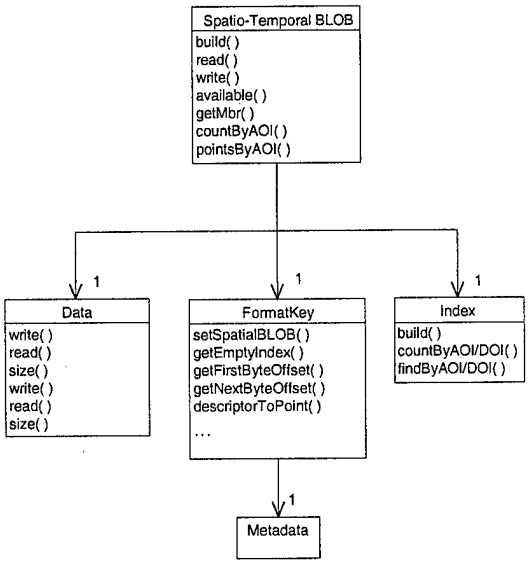

Figure 1. An object decomposition of the Spatio-Temporal BLOB

The above diagram shows that the Spatio-Temporal BLOB is actually composed of four distinct subcomponents. We will describe the specific roles of each of these components in accomplishing the overall purpose of the Spatio-Temporal BLOB.

Data: Of course, the principal responsibility of the repository is to hold data. The Data component borrows functionality directly from standard BLOBs as implemented in Ozone. Binary data is stored in a series of moderate sized pages, each implemented as a byte array. Data is stored and retrieved according to byte offset. Each data point is represented as a binary record within the data.

Format Key: The Format Key's primary role is to translate individual byte records into object representations of that same data. When the Spatio-Temporal BLOB fulfills requests for data by area-of-interest or duration-of-interest the Format Key is given the offsets of relevant byte records. It reads the record, interprets it, and constructs a corresponding point object. Its more general role is to capture all information about a specific internal data format[2].

Index: The Index maintains a mapping between the spatial or temporal position of a data point and its byte offset within the Data. When the Spatio-

[2] This includes knowledge concerning the positions of the records such as the offset of the first record, the position of one record given the position of the previous, etc. This functionality is used for indexing the data. We mention it but will not discuss it in detail.

Temporal BLOB receives a request for data according to an area-of-interest or duration-of-interest it is the Index's responsibility to determine the offset of all matching records.

Metadata: We have already identified the principal components of the Spatio-Temporal BLOB. However, to make our picture complete we must add one more. Many binary formats for meteorological data are not fully self-descriptive and, therefore, require additional metadata to fully interpret their contents (we will address this issue in more detail when we discuss the Spatio-Temporal BLOB as an abstract framework).

The Spatio-Temporal BLOB, itself, performs an orchestrating role, forwarding requests to the subcomponents and returning results to the client. Let us look at how it works together with its subcomponents to fulfill a request for data:

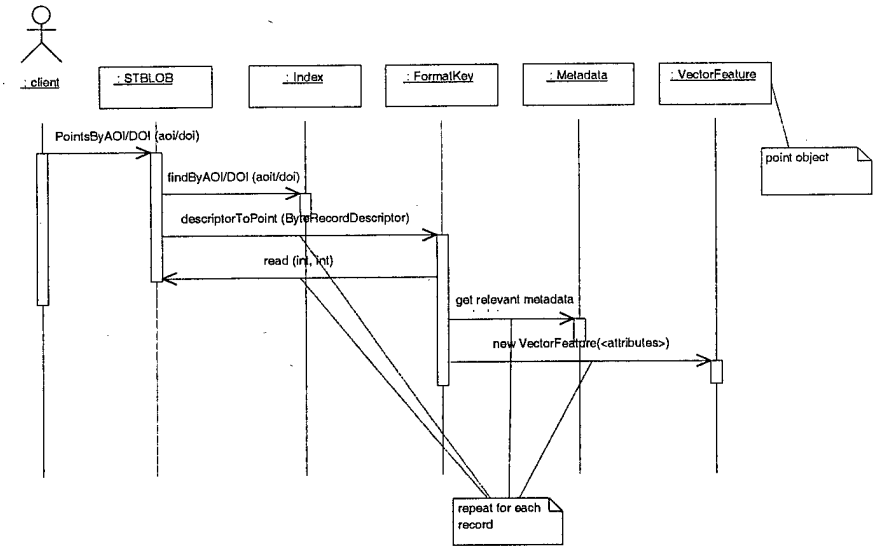

Figure 2. The sequence of steps to retrieve data from a Spatio-Temporal BLOB by area-of-interest or duration-of-interest

Typical data retrievals have the following steps:

1. The client requests the points contained within an area-of-interest or duration-of-interest from the Spatio-Temporal BLOB.
2. The Spatio-Temporal BLOB forwards the request to the index that returns an array of record descriptors, each specifying a record's byte offset and the spatial or temporal position of the point it describes.

3. The Spatio-Temporal BLOB iterates through the descriptors, passing them in turn to the Format Key that reads the appropriate record from the data and constructs a point object embodying the information.
4. The Spatio-Temporal BLOB returns the resulting array of points.

8. A FLEXIBLE FRAMEWORK

We have identified the fundamental subcomponents of the Spatio-Temporal BLOB, but Spatio-Temporal BLOB framework does not actually define concrete implementations for all of them. Instead, for certain subcomponents (the Index and Format Key), the Spatio-Temporal BLOB framework only specifies an interface that indicates what functionality any actual implementation must provide. Different implementations will be used depending on the binary format of the Spatio-Temporal BLOB's contents.

Suppose, for example, that we wish to allow the Spatial BLOB to store gridded meteorological data. To accomplish this we extend the framework to handle data stored in the GRIB format, a format commonly used for the storage and transfer of gridded meteorological data [WMO 2002]. An implementation of the Format Key must be provided which encapsulates all of the knowledge of how to interpret the GRIB data format. When the Spatial BLOB is built and populated with GRIB data this key is plugged in, giving the Spatial BLOB the knowledge of how to access and translate its contents. Likewise, an implementation of the Index interface should be chosen which is well suited to GRIB data (choosing the best index is generally dependent on which has the best performance with that type of data). Also, the Spatial BLOB must be initialized with appropriate Metadata attributes describing the specific data set which the Spatial BLOB contains. The particular set of attributes depends on the particular format being used. Required metadata for describing GRIB data includes, among other things, information about the structure of the grid for that data set. If one GRIB BLOB contains data describing a data set with a grid of dimensions 100 x 500 with intervals of 20 kilometers, the Metadata specifies this information. A different GRIB BLOB containing a different data set with different dimensions would have metadata specifying its particular dimensions. The same holds true for Temporal Blobs. An appropriate Temporal Format Key will be plugged in depending on the format of the temporal data. The appropriate index implementation would vary depending on whether the contents described events at evenly spaced or irregularly spaced times. Again, the appropriate set of metadata parameters depends on the type of data we are dealing with. We will look at each subcomponent in turn to see if and how it varies.

Data: The implementation of Data remains constant, regardless of the data type contained by the Spatio-Temporal BLOB. The reason is simple: all of the data we are dealing with, regardless of its particular format, is binary. The Data component is simply a storage mechanism for binary data, and the semantics of the data it contains is irrelevant.

Format Key: Clearly the process of interpreting the data varies from format to format. We will provide different implementations of the Format Key for each format we handle. When the Spatio-Temporal BLOB is built it is initialized with a particular Format Key implementation that is appropriate for its contents. In addition to encapsulating knowledge of the structure of the data format, the key also specifies which implementation of the Index interface will be used for that format. The key does this by implementing a factory method `getEmptyIndex()` which returns the appropriate index implementation. When the Spatio-Temporal BLOB is being built, it calls this method and receives the appropriate type of index without ever knowing which implementation it is actually using.

Index: There are a wide variety of well-known indexing schemes available. GIDB, for example, uses an R*-tree as its primary index. Any implementation can be used as long as it provides the functionality specified by the Index interface. However, not all schemes are ideally suited to all data types. Again, we consider the differences between gridded and irregular spatial data. While many indexing schemes are sufficiently general to handle all of the data configurations we anticipate, this generality is expensive. An R*-tree may be suitable for irregularly spaced data points, but simpler and more efficient alternatives exist for gridded data. For gridded data we use a "Computational" implementation of the index that is not a typical index in the sense of having entries added and maintained in a searchable structure. It simply uses a straightforward linear equation to compute on-the-fly what points in the grid match an area-of-interest and what the byte indices of the corresponding records are. Since no entries are stored no storage space is required. For large, gridded data sets this results in enormous savings. Similarly, for regular data Temporal BLOBs use a Temporal Computational Index that computes timestamps with a simple linear calculation using the start time and regular interval between events.[3]

Metadata: The metadata attributes that fully qualify a data set of a given type vary according to that type. When the Spatio-Temporal BLOB is built it is initialized with a metadata appropriate for its contents.

[3] The best means of indexing irregular temporal data within the GIDB System is still being evaluated. A variation of an R*-tree which treats time in a similar manner to other dimensions is being considered.

Since format specific components must be plugged in before meaningful data can be retrieved, the building of the Spatio-Temporal BLOB is a multi-step process. The details are as follows:

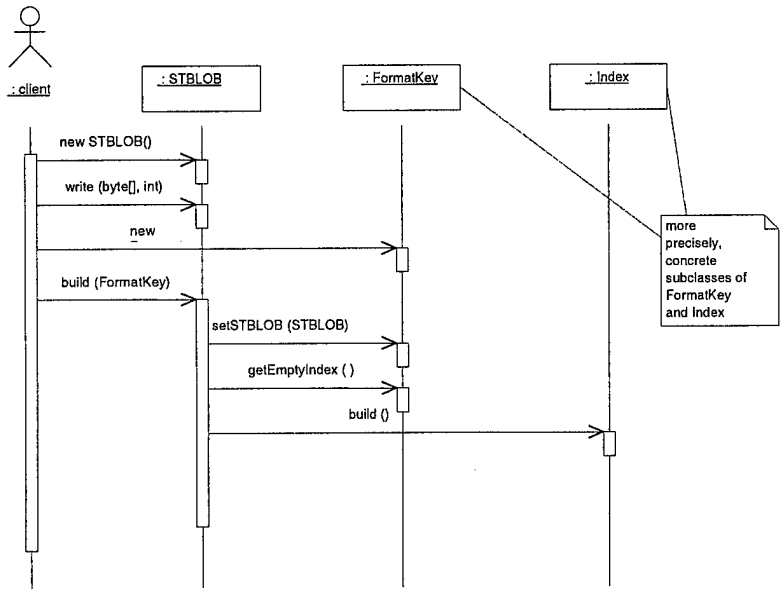

Figure 3. The sequence of steps to build a Spatio-Temporal BLOB

1. Construct an empty Spatio-Temporal BLOB.
2. Write the binary data to the Spatio-Temporal BLOB (typically this data is written as is, in its original file format; when appropriate it may be pre-processed to convert it into some more suitable from).
3. An appropriate subclass of Format Key is constructed with appropriate metadata for the specific data set contained in the Spatio-Temporal BLOB.
4. The Spatio-Temporal BLOB is built by initializing it with the key. This is the step that fully defines the Spatio-Temporal BLOB, providing it with the "knowledge" of the semantics of its contents. This step involves sub-steps:
   a) The Spatio-Temporal BLOB passes itself as a reference to the key.
   b) The Spatio-Temporal BLOB retrieves an empty index of the appropriate implementation from the Format Key and then populates the index.

9. SAMPLE APPLICATIONS

To understand the frameworks better we will look at how they are applied to actual specific formats. First, we will look at the Spatial BLOB extension for the GRIB format.[4] Each GRIB file describes the state of the atmosphere for some parameter (we will only be dealing with files which describe a single parameter although GRIB may handle multiple parameters per file). It contains records specifying the values for that parameter at a series of gridded points over that area. Each GRIB record is a single four-byte real number representing the value for the given parameter at the corresponding point. The points are arranged in row-prime ordering with x and y increasing in the positive direction.

The principal task required in extending the Spatial BLOB framework to the GRIB format is to define a GRIB Key implementation of the Format Key interface. This key will do four things:

1. Specify the set of required Metadata attributes. These will be taken as arguments to the GRIB Key constructor. The specific attributes required for GRIB data are the parameter being modeled (temperature, precipitation, etc.) and information about the grid including the number of rows and columns, the distance between points, the mapping projection used, and any additional values needed to describe a specific projection.
2. Specify which implementation of the Index interface will be used. Since GRIB data is gridded, the Computational Index is an appropriate choice. The GRIB Key's `getEmptyIndex()` method constructs and returns a new Computational Index.
3. Provide functionality to convert the byte records to point objects. Because the GRIB format is very simple, implementing this functionality is straightforward. It simply accesses the record at the specified offset, reads in the value, and builds a point feature object with the associated latitude and longitude and a single attribute consisting of the attribute name and value.
4. Provide auxiliary functionality describing the positions of the records. Since the first record begins at zero and all records are four bytes, providing this information is simple.

The implementation of the GRIB Key is straightforward, allowing for an efficient and easy application of the Spatial BLOB framework. The GRIB

[4] Actually, the GRIB file format, itself, is relatively complex. It includes a large amount of header metadata and uses compression. We will not actually be dealing with GRIB but with an unpacked version of GRIB. For convenience's sake, however, we will refer to this as GRIB data.

Key has been implemented and deployed for regular use in the GIDB System. In the next section we will evaluate its success in meeting our requirements.

We will briefly sketch a similar example for the Temporal BLOB. Suppose we have an observation station that samples an atmospheric parameter such as temperature at regular intervals. To allow the Temporal BLOB to store this data we will define a very simple binary temporal data format. This format will store the value of the parameter for each point in time as a four-byte float, just like the GRIB example does for each point in space. Since the format is describing events at regular time intervals, we can handle this in the same way our previous example handled spatial grids. Instead of storing a specific time stamp with each record, we store metadata describing the time intervals. The complete set of metadata parameters necessary to describe the data is:

1. The parameter being described (i.e. "Temperature").
2. The units (i.e. "Celsius").
3. The time of the first observation.
4. The interval of time between observations.

To be able to store data of this format in a Temporal BLOB we must define an appropriate key, which we will call a Simple Float Key. Its functionality is similar to that of the GRIB Key. It specifies the required metadata attributes, determining which index implementation is used, and provides functionality to extract the appropriate data and return it in the desired form.

10. EVALUATION

Now that we have described the design and behavior of our solution in detail, we will see how it compares to the criteria specified in our requirements. For a test, we will evaluate the quantitative performance of the Spatial BLOB, measuring its behavior with a typical GRIB data set. The test data set has the following characteristics:

- The data set is output from the Coupled Ocean/Atmosphere Mesoscale Prediction System (COAMPS) meteorological model's Continental U.S. run [COAMPS 2001]. This particular output data set describes temperature.
- The data set is a grid with 301 columns and 131 rows (39,431 points total). The bounding box of the data set is: 126W, 24N, 66W, 50N.

The following tests compare the performance of the Spatial BLOB containing this data as opposed to the same data stored using the previously described naïve object approach (each point modeled as an object).

The first criterion we will examine is the storage size of the data with the different methods. The following table shows the sizes of the data stored in its original flat file format, in the Spatial BLOB, and as point objects:

Table 1. Storage size of the data in bytes

Original flat file	Spatial BLOB	Point objects
54,302 (53.0 KB)	66,652 (65.0 KB)	4,712,991 (4.5 MB)

The size of the Spatial BLOB is comparable to that of the original GRIB file[5] and is many times smaller than the point object representation. The reasons for this are twofold:

1. The primary reason, as previously mentioned, is that the binary representation of the data is more compact than the object representation.
2. The Spatial BLOB gains an additional advantage in storage size by its use of the Computational Index for gridded data. Since this "index" computes the offsets of records on the fly, no storage space is required.[6]

Clearly, the Spatial BLOB provides a very large improvement in storage size over the naïve object approach. The concern is that this improvement is gained at the expense of excessive access times since data must be converted on the fly from binary to object form. Testing reveals that this is, in fact, not the case.

Table 2. Time to retrieve objects in milliseconds (various areas-of-interest)

Query area	Spatial BLOB	Point Objects
96W, 37N, 95W, 38N (small - 36 points)	140	2,544
126W, 37N, 66W, 38N (wide - 1,806 points)	431	911
96W, 24N, 95W, 50N (tall - 786 points)	80	872
126W, 24N, 66W, 50N (all - 39,341 points)	4,626	107,835

Retrieving data from the Spatial BLOB using the Computational Index implementation is significantly faster than retrieving point objects using the database's R*-tree index, thus, more than compensating for the relatively small amount of time needed to convert the data to object form.

The Spatial BLOB outperforms the naive object solution across the board.[7] The binary storage of the data provides a massive reduction in storage size. The minimal cost that is incurred by the on-the-fly conversion

[5] Both GRIB and Ozone compress data. In this case, the Spatial BLOB is slightly larger than the original GRIB file. Experience with other, non-compressed file formats shows that the Spatial BLOB is often substantially smaller than the original file.

[6] Approximately 4 MB of the naïve solution's size is actual storage of data. The additional 0.5 MB of size is the space needed for the index.

[7] In fact one performance benefit of the Spatial BLOB that was not an explicit objective of our requirements is significantly reduced build times. For the test dataset it took 3,124 milliseconds to build as a Spatial BLOB versus 105,702 milliseconds with the naïve object approach.

of the data is more than offset by the flexible indexing scheme that can provide benefits in both storage size and access times. Our performance requirements have been met, and in some cases exceeded (we required that our solution provide an improvement in storage size without an excessive sacrifice in access times; our solution provides, at least in some cases, an improvement in storage size and an *improvement* in access times).[8]

The Temporal BLOB is still in its prototype stage and will be fully incorporated into the GIDB System as the temporal aspects of the system are extended. Currently, no benchmarking of its behavior is available, but, since it is fundamentally similar to the Spatial BLOB, its performance is expected to be comparable.

11. FUTURE DEVELOPMENTS

Spatial and Temporal BLOBs were designed to handle specific, real-world needs for storage of anticipated data sets. They meet those needs well, but their uses are specialized. They respectively handle horizontal, two-dimensional areas at a single time and single points at many times. A natural next step is to develop a fully generalized, four-dimensional, true Spatio-Temporal BLOB capable of describing the environment in three spatial dimensions plus time. This new solution would be able to handle all of the data sets stored by the two current solutions as well as others that neither of them can store. For example, currently, to describe the temperatures in the atmosphere, multiple Spatial BLOBs store multiple data sets describing stacked, horizontal slices of the atmosphere at various altitudes. To store temperature variations over time data sets are required for each time as well as each altitude. This solution reflects the way most atmospheric models currently output results. However, the division of data into multiple altitudes and times is arbitrary. These divisions do not allow the data sets to express relations between points in different sets. The single slice can only express proximity between points along the horizontal axes but not the vertical or temporal axes. The proposed integrated Spatio-Temporal BLOB would be able to represent all of this information in a single data set. Similarly, data in a Temporal BLOB can express proximity in time but not in any other

8 The benchmarks described above are for the application of the Spatial BLOB framework to gridded data where the Spatial BLOB's superiority over the naive object approach is greatest. This is because the gridded data can be indexed using the Computational Index implementation, which is fast and requires no storage. Irregular data will require some other scheme, which will have costs in storage size and/or speed, but these costs are no greater than that of any other solution. The flexibility of the Spatial BLOB allows for the choice of the best possible indexing scheme.

direction. Again, the integrated Spatio-Temporal BLOB would be able to express these relations. As it stands, an application using the multiple, separate data sets for analysis purposes must reconstruct a unified picture of the environment in order to capture these relations. A unified Spatio-Temporal BLOB would explicitly capture this information.

In developing an integrated Spatio-Temporal BLOB the basic architecture would be nearly identical to that of the existing solutions. The principal difference would be in the adoption of appropriate binary formats and indexing schemes for four-dimensional data.

12. CONCLUSION

By resolving the technical difficulties presented by the incorporation of large volume meteorological data into an object-oriented database, the Spatial and Temporal BLOB frameworks provide the fast access to compactly stored, meaningful data required to facilitate the GIDB System's use as a planning aid and analysis tool. The Spatial BLOB has been deployed in the GIDB System and has proved to be an operational success. The Temporal BLOB holds similar promise.

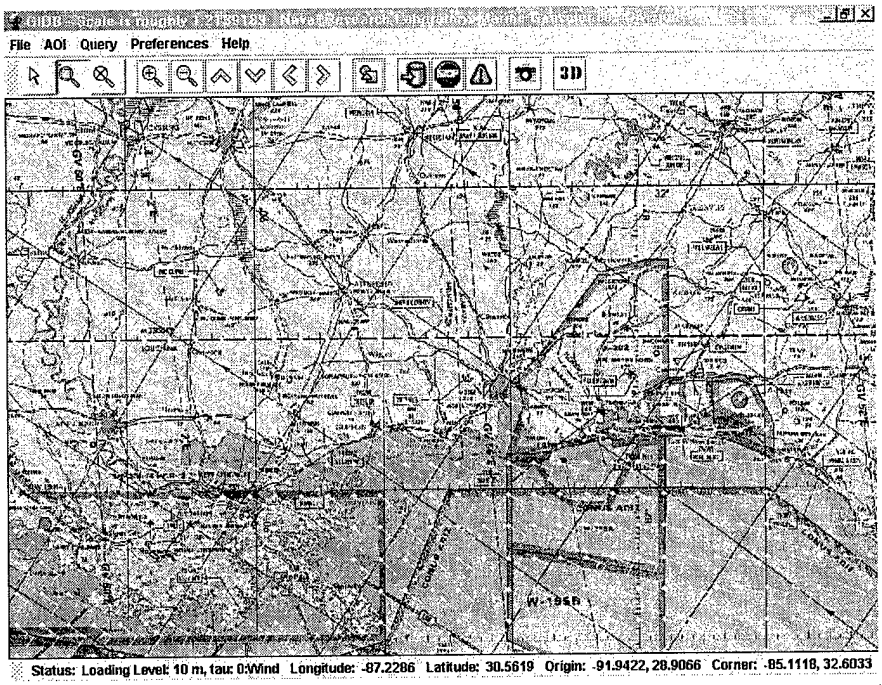

Figure 4. Data from Spatial and Temporal BLOBs may be used both for visualization and analysis purposes. This picture shows a view of wind data extracted from a Spatial BLOB and superimposed on a raster background map.

ACKNOWLEDGEMENTS

We would like to thank the Naval Research Laboratory's Base Program, (Program Element 0602435N) and also the Office of Naval Research's Generation and Exploitation of the Common Environment (GECE) Program (Program Element 63782N).

REFERENCES

[COAMPS 2001] Naval Research Laboratory Monterey Marine Meteorology Division. Coupled Ocean/Atmosphere Mesoscale Prediction System. 19 Feb. 2001 <http://www.nrlmry.navy.mil/projects/coamps/>.

[NRL 2002] Naval Research Laboratory Digital Mapping, Charting and Geodesy Analysis Program. DMAP Home Page. 4 Mar. 2002 <http://dmap.nrlssc.navy.mil/>.

[Owen and Voges 1997] James Owen, Erik Voges. A Generic Indexing Mechanism for Persistent Java. Sept. 1997 <http://people.cs.uct.ac.za/~evoges/web/Paper/p.html>.

[Ozone 2002] Ozone Database Project. ozone – The Open Source Java ODBMS. Apr. 2002 <http://www.ozone-db.org/ozone_main.html>.

[WMO 2002] World Meteorological Organization. Part II, A Guide To The Code Form FM 92-IX Ext. GRIB, Edition 1. Apr. 2002 <http://www.wmo.ch/web/www/WDM/Guides/Guide-binary-2.html >.